THE UNIVERSITY OF MELBOURNE

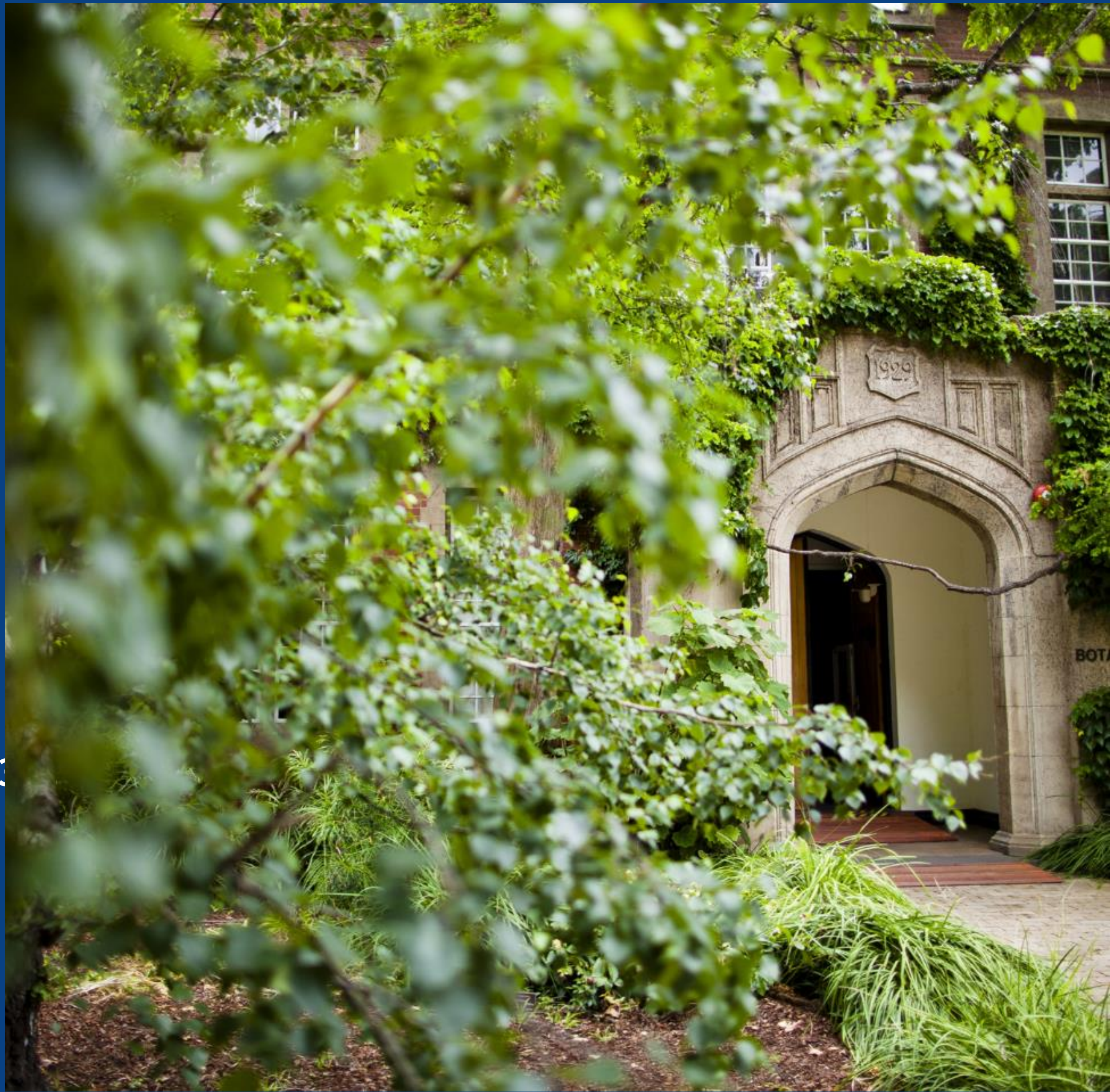
COMP 90048 Declarative Programming Workshop 7 (week8)

2019 semester 1

by Wendy Zeng

Tutorial : Tue 18:15 - 19:15 221 Bouverie St, room B11

Wed 17:15 - 18:15 201 Bouverie St, room B132



Outline

1. Tail Recursion Optimization in Prolog
    a. Factorial
    b. Prolog List: Reverse
2. Difference Lists
    a. Prolog Tree: Tree to List
    b. Prolog List: QuickSort
3. Some clarifications on common operators:

    =, ==, =:= , is

1. Tail Recursion Optimization in Prolog

- **Tail Recursion Optimization:**
  - When the recursive call is the last function invoked in the evaluation of the body of the function.
  - Can be implemented in many programming languages besides Prolog
- Motivation for Tail Recursion:
  - Efficiency: **consumes less stack depths** compared to full recursive functions. Intermediate call stacks are not needed to resume when recursion returns with values. The last recursive call can be returned straight to the initial call stack.
  - Tail recursive functions are a mimic of **loop** in imperative function
- Usually achieved with the use of accumulator

1. Tail Recursion Optimization in Prolog

Full Recursion:

```
factorial1(N, F) :-
    ( N =:= 0 ->
            F = 1
    ; N > 0 ->
            N1 is N-1,
            factorial1(N1, F1),
            F is F1*N
    ).
```

Tail Recursion:

```
factorial2(N, F) :- factorial2(N,
1, F).
factorial2(N, A, F) :-
    ( N =:= 0 ->
            F = A
    ; N > 0 ->
            N1 is N-1,
            A1 is A*N,
            factorial2(N1, A1, F)
    ).
```

A: accumulator

- Has an "initial" value (mostly [] or 0 or 1)
- Holder of temporary computation result up to current recurssion step (etc. up to number of N)
- New value of accumulator is computed (A1), then passed down into the recurssion

1. Tail Recursion Optimization in Prolog

Full Recursion:

```
factorial1(N, F) :-
    ( N =:= 0 ->
            F = 1
    ; N > 0 ->
            N1 is N-1,
            factorial1(N1, F1),
            F is F1*N
    ).
```

F = 1
F = 1 * 1
F = (1 * 1) * 2
F = ((1 * 1) * 2) * 3

Tail Recursion:

```
factorial2(N, F) :- factorial2(N,
1, F).
factorial2(N, A, F) :-
    ( N =:= 0 ->
            F = A
    ; N > 0 ->
            N1 is N-1,
            A1 is A*N,
            factorial2(N1, A1, F)
    ).
```

A = 1
A = 1 * 3
A = (1 * 3) * 2
A = ((1 * 3) * 2) * 1
When N =:= 0, F = A

1. Tail Recursion Optimization in Prolog

Naive Reverse:

```
reverse1([], []).
reverse1([Head|Tail], Rev) :-
    reverse1(Tail, Rev_tail),
    append(Rev_tail, [Head], Rev).
```

Result Rev is built up on the way up the recursive stacks

Tail Recursive Reverse:

```
reverse2(List, Rev) :- reverse2(List, [], Rev).
reverse2([], Acc, Acc).
reverse2([H|T], Acc, Rev) :-
    reverse2(T, [H|Acc], Rev).
```

Return the accumulator result

Accumulate result down

Rest part of the list that remains to be reversed

Partial reversal of the previous part of the list

Final reverse of the entire list

Result Rev is built on the way down the recursive stacks

1. Tail Recursion Optimization in Prolog

Which of the following function is tail recursive?

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x):(map f xs)
```

(:) is executed after the recursion call

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs) =
  if f x then x:fxs else fxs
  where fxs = filter f xs
```

(:) is executed after the recursion call

```
foldl :: (v -> e -> v) -> v -> [e] -> v
foldl _ base [] = base
foldl f base (x:xs) =
  let newbase = f base x in
  foldl f newbase xs
```

```
foldr :: (e -> v -> v) -> v -> [e] -> v
foldr _ base [] = base
foldr f base (x:xs) =
  let fxs = foldr f base xs in
  f x fxs
```

f is applied after the recursion call

2. Difference Lists

What is Difference Lists:

- Knowing the structure of a list up to a point
- The remaining of the list can be left unbound until the complete evaluation of a predicate

- 1. Prolog tree: Tree to List

  Naïve version:

```
tree_list1(leaf, []).
tree_list1(node(L, V, R), List) :-
      tree_list1(R, R_list),
      tree_list1(L, L_list),
      append(L_list, [V|R_list],
List).
```

Using Accumulator:

```
tree_list2(Tree, List) :- tree_list2(Tree, [],
List).

tree_list2(leaf, List, List).
tree_list2(node(L, V, R), List_in, List_out) :-
      tree_list2(R, List_in, List_R),
      tree_list2(L, [V|List_R], List_out).
```

- Post order tree traversal (right to left)
- The accumulator is a temporary computation result that holds all the elements from nodes that have already been visited

2. Difference Lists

- 1. Prolog tree: Tree to List

Using Difference Lists:

```
tree_list3(Tree, List) :- tree_list3(Tree, List, []).

tree_list3(leaf, List, List).
tree_list3(node(L, V, R), Full_list, End_of_list) :-
    tree_list3(L, Full_list, End_of_list_L),
    End_of_list_L = [V|List_R],
    tree_list3(R, List_R, End_of_list).
```

tree_list3(+Tree_of_elem, -List_of_all_elem, +Elems_to_be_at_the_end)

Full_list: is a list of all elements from L with End_of_list_L appended to it's end

What should be appended to the end of left list is decompmosed in to the current value in node plus everything from the right branch

List_R: is a list of all elements from R with End_of_list appended to it's end

2. Difference Lists

- 2. Prolog List: QuickSort

Naïve QuickSort:

```
greater(Pivot, X) :- X>Pivot.

quicksort1([], []).
quicksort1([H|Tail], Sorted) :-
    partition(greater(H), Tail, Larger, Smaller),
    quicksort1(Smaller, LeftList),
    quicksort1(Larger, RightList),
    append(LeftList, [H|RightList], Sorted).
```

- Similar to Haskell's version of QuickSort:

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =   quicksort smaller ++ [x] ++ quicksort larger
    where
    smaller = [ t | t <- xs, t<x ]
    larger  = [ t | t <- xs, t>=x ]
```

2. Difference Lists

- 2. Prolog List: QuickSort

Using Difference Lists:

```
quicksort2(List, Sorted) :- quicksort2(List, Sorted, []).

quicksort2([], Rest, Rest).
quicksort2([Pivot|Tail], Sorted_all, End_of_sorted_all) :-
    partition(greater(Pivot), Tail, Larger, Smaller),
    quicksort2(Smaller, Sorted_all, End_of_sorted_small),
    End_of_sorted_small = [Pivot|Sorted_large],
    quicksort2(Larger, Sorted_large, End_of_sorted_all).
```

quicksort2(+Unsorted_list, -Sorted_list, +Already_sorted_end_of_list)

3. Some clarifications on common operators:

=: Unification

- *What it does:* unify terms on both sides of the sign
- *When it succeeds:* when terms on both sides are unifiable
- \=

==: Term comparison

- *What it does:* test whether terms on both sides are literally identical
- *When it succeeds:* identical terms
- \==, @<, @=<, @>, @>=
- Term comparison rule:
  Var < Number < String < Atom < Compound Term

3. Some clarifications on common operators:

is: Arithmetic Evaluation

- *What it does:* evaluate mathematical expression on right hand side and unify it with the unbound term on the left
- *When it succeeds:* left hand side is unbound and right hand side is proper mathematical expression

=:= Arithmetic Comparison

- *What it does:* evaluate expressions on both sides and compare the results
- *When it succeeds:* evaluation results from both sides are literally identical
- =\=, <, >, =<, >=

THE UNIVERSITY OF MELBOURNE

Thank you

wendy.zeng@unimelb.edu.au

By Wendy Zeng